J. Symbolic Computation (1993) **15**, 641–672

QPC_2 : A Constructive Calculus with Parameterized Specifications

Yukihide Takayama
Department of Computer Science and Systems Engineering
Ritsumeikan University
56-1, Kita-machi Tohjiin, Kita-ku, Kyoto 603, Japan
email:takayama@cs.ritsumei.ac.jp

December 28, 1992

Abstract

A second order constructive calculus is presented in this paper. The idea is not to give a yet another system but to restrict higher order calculi such as the Calculus of Constructions (CoC) [2, 3] and Martin-Löf's Intuitionistic Type Theory (ITT) with the hierarchy of universes [10]. One of the aims of the restriction is to provide a program synthesis system which directly generates functional programs with the program constructs such as if-then-else and pairing. Unlike CoC, F [5], and F_ω [4], we do not use Prawitz coding of logical connectives [15] because, if we use the coding, the constructs such as if-then-else and pairing are not primitives but the functionals defined in higher order lambda terms. Another aim of the restriction is to find a subset of existing higher order systems which is necessary and sufficient for parameterized specifications. The obtained system, which is called $\mathbf{QPC}_2$, roughly corresponds to a subset of ITT with the universes U_1 and U_2, but has a few differences. First of all, unlike ITT $\mathbf{QPC}_2$ allows universal quantification over predicates which enables a flexible description of parameterized specifications. Secondly. we take type-free approach along the line of PX [8] and SST [16] in which types and formulas are clearly separated and they are linked with realizability interpretation. Thus, each universe is separated into two parts: one for types and another for formulas. This distinction of types and formulas combined with the realizability technique enables the program extraction without redundancy. A program scheme is extracted from the proof of a given parameterized specification, and it is reduced to a program when first order formulas are substituted to parameters. Because of the suitable restriction of second order formulas, most part of the instantiation procedure can be easily performed automatically. Also, $\mathbf{QPC}_2$ together with the various optimization techniques for first order constructive calculus such as the extended projection method [19] can synthesize natural programs.

0747–7171/93/5–6641 + 32 $08.00/0

1 Introduction

It is well known that formal development of functional programs can be carried out in constructive logics, and there are basically two kinds of formalism of the logics. One is the formalism of constructive type theories such as Martin-Löf's Intuitionistic Type Theory (ITT) and the Calculus of Constructions (CoC), and another is the formalism of intuitionistic logics with realizability interpretations [22] such as PX. We call the latter *type free formalism* in the following. To gain high level expressive power, most constructive type theories are designed as higher order systems which can represent higher order logics. On the other hand, there are few higher order systems in the type free formalism designed as a framework for the formal development of functional programs.

In the higher order type theories such as F, F_ω and the CoC, logical connectives are not primitive constructs but the functionals defined by Prawitz coding [15]. Although Prawitz coding makes formulation of higher order calculi very simple preserving the expressive power, the use of Prawitz coding raises a problem in program synthesis. Natural programs with the primitive constructs such as if-then-else and pairing can be directly synthesized in the first order constructive systems. But if the logical connectives are defined by Prawitz coding, the constructs are not primitive. For example, if-then-else construct is in the form of $tC(\lambda x^A.u)(\lambda y^B.v)$ where t is either $\Lambda X.\lambda p^{A\to X}.\lambda q^{B\to X}.pr$ or $\Lambda X.\lambda p^{A\to X}.\lambda q^{B\to X}.qs$ for suitable types A, B and C. This means that we need a higher order typed calculus to describe the extracted programs. It will be possible to introduce the primitive logical connectives into those higher order systems, but the obtained systems will be rather complex and redundant. On the other hand, Martin-Löf's ITT does not have this kind of problem because it does not use Prawitz coding and programs with the primitive constructs can be extracted in the first order fragment.

Although ITT has an infinite cumulative hierarchy of universes, the second order fragment in the second universe seems to be almost sufficient for ordinary programming because abstract data type definitions and parameterized module definitions are possible in this fragment as demonstrated in [13]. A little more observation tells us that the usage of the second order universal quantifier is restricted: the universal quantifier does not occur inside the definition of a parameterized module. Therefore, it seems that a restricted version of ITT with the universes U_1 and U_2 is almost sufficient for programming with parameterized modules. However, ITT has the following problems. First of all, because of the lack of elimination rules with regard to the universes, the type expressions such as $(\Pi\alpha \in U_1)(\Pi P \in \alpha \to \alpha \to U_1)(\Pi x \in \alpha)(\Sigma y \in \alpha)(P\ x\ y)$ are not allowed in the original version of ITT. This means that, in specifying parameterized modules, we can only quantify over propositions and universal quantification over predicates are not allowed. The second problem is that identification of types and proposition in ITT is not always suitable for program synthesis in the following sense. It is not always necessary to give the computational meaning to all the types in the program extraction because some of the types are only used for describing the logical properties of the programs, and any programs should not be extracted from the proofs of this kind of types. This observation led the idea of the subset types in ITT [1, 13] and the distinction of informative types and non-informative types in CoC [14] to remove the redundancy in the extracted programs.

We present in this paper a second order constructive calculus called $\mathbf{QPC_2}$. $\mathbf{QPC_2}$ does not use Prawitz coding of logical connectives since we aim to synthesize natural programs with the primitive constructs. The calculus is roughly a restriction of Martin-Löf's ITT with universes U_1 and U_2, but it allows quantification over predicates which improves the expressive power of the calculus in describing the parameterized specifications. To remove the redundancy in the extracted programs, we use the extended projection method (EPM) [18, 19]. EPM works well in the first order calculi in type free formalism allowing more fine grained semi-automatic analysis of redundancy than the subset types and the informative/noninformative type system. Thus, we formalize $\mathbf{QPC_2}$ as a type free system.

Unlike ITT, $\mathbf{QPC_2}$ does not have the second order existential quantifier corresponding to the strong sum which enables the module specification. Therefore, our system is a second order calculus for parameterized specifications, not modules. The second order formulas of $\mathbf{QPC_2}$ are restricted in the sense that the second order universal quantifier does not occur inside a formula. This restriction makes the proof normalization procedure with regard to the quantifier drastically simple. This is a nice property from a practical viewpoint because it makes automatic instantiation of the parameterized specifications very easy.

The organization of the paper is as follows. Section 2 briefly gives the definition of the target programming language λ^{qpc} which is a variant of untyped lambda calculus with the primitive constructs. The type system of $\mathbf{QPC_2}$ is presented in section 3. The types are used both for typing the synthesized codes and for specifying the domains of the quantifiers. The synthesized codes are not always the programs. Codes with the parameters will be synthesized from proofs of the parameterized specifications. So that there is a distinction of programs and *program schemes* in the synthesized codes. This distinction is reflected in the type system of $\mathbf{QPC_2}$. Namely, there are three kinds of subsystems: subsystems for programs, program schemes and the domains of the second order quantifier. Here we see another reason for the type free formalism of $\mathbf{QPC_2}$. We aim to compromise the type system of existing typed functional languages such as Standard ML. Our calculus, λ^{qpc}, is not exactly any existing language but is very close to them because it is an ordinary untyped λ-calculus typed by a simple type system. A type free system can naturally incorporate such an exotic type system in it. The precise definition of the program schemes is given in section 4. A program scheme almost looks like a program but some special constructs containing predicate variables may occur in it. A program scheme is converted to a λ^{qpc} term when the predicate variables in it are instantiated to suitable first order predicates. The second order rules and some properties of $\mathbf{QPC_2}$ proofs are also presented in section 4. Section 5 presents the full detail of the code extraction from proofs. A variant of q-realizability interpretation of $\mathbf{QPC_2}$ is defined. The soundness proof of the interpretation gives the program extraction algorithm. Examples of program definitions and extractions are given in section 6 and 7. Final remarks are given in section 8.

In the following, substitution of an expression, T, to a variable (or sequence of variables), X, which occurs free in an expression, E, is denoted $E_X[T]$. If X is a sequence of variables then T must also be a sequence of the same length. $E_{x_1,\ldots,x_n}[M_1,\cdots,M_n]$ denotes simultaneous substitution. If A is a formula, $A_x[M]$ is also denoted $A(M)$.

2 The Target Programming Language: λ^{qpc}

λ^{qpc} is the term calculus of QPC_2. It is essentially an ordinary untyped lambda calculus with a fixed point operator, if-then-else and pairing. In order that the realizability can be easily formulated and the extended projection method [19] can be directly applied, λ^{qpc} has slightly nonstandard syntax.

- constants: natural numbers, $left$, $right$, any, nil, T (true), and F (false)
- individual variables: x, y, z, $\cdots$
- lambda abstraction: $\lambda x.M$
- application: $ap(M, N)$
- constructors: :: (list constructor), if-then-else, $\mu z.M$ (fixed point operator), let-sentence,
- sequence of terms: $(M_1, \cdots, M_n)$, () (empty sequence)
- primitive functions: hd (head of lists), tl (tail of lists), $beval$, $succ$ (successor), $pred$ (predecessor), app (append function), and those for handling sequences of terms: $tseq$, $ttseq$ and $proj$.

$beval(\ )$ is a primitive function which performs the decision procedure of equalities of terms, and $if\ A\ then\ M\ else\ N$ is an abbreviation for $if\ beval(A)\ then\ M\ else\ N$. Notice that as μ-terms are allowed, the equalities of the terms are not always decidable. Particularly, an equation containing the if-then-else constructs is not generally decidable because the equation part, A, in $if\ beval(A)\ then\ M\ else\ N$ may contain a μ-term. $tseq$, $ttseq$ and $proj$ are defined as follows:
$tseq(k)(\overline{s}) = (s_k, s_{k+1}, \cdots, s_n) \quad (1 \leq k \leq n)$
$ttseq(k,l)(\overline{s}) = (s_k, s_{k+1}, \cdots, s_{k+l-1}) \quad (1 \leq k \leq n, 1 \leq l \leq n-k+1)$
$proj(k)(s_1, \cdots, s_n) = s_k \quad (1 \leq k \leq n)$
As the notation, the sequences are often denoted $\overline{M}$ and $\overline{x}$ (sequence of variables). The special sequence of the constant any of length n is denoted $any[n]$. Notice that the variables used in lambda abstractions, let-sentences, and the fixed point operator can also be the sequences of variables. They are often denoted in upper case letters, X, Y, Z, $\cdots$.

We use the sequences of terms to represent the realizers of the conjunction and the disjunction formulas. A sequence can be understood as a mixture of pairing (a, b), $i(a)$, and $j(b)$ notation in Martin-Löf's ITT. $i(a)$ and $j(b)$ terms in ITT are represented by the sequences $(left, a, any[n])$ and $(right, any[m], b)$ for suitable natural numbers n and m. The sequences $any[m]$ and $any[n]$ are necessary for formalizing the typed realizabiliy, and a sequence of any represents a realizer of the formula in a disjunction which is not actually proved.
The basic idea of the extended projection method is to represent realizers as sequences of terms and to give an algorithm to extract, by analyzing proof trees, particular subsequences of the realizer as the redundancy free programs. Thus, the sequence syntax is heavily used in the method.

$M \rhd N$ means that the term M is reduced to N.

$$\frac{M_1 \rhd N_1 \quad M_2 \rhd N_2}{(M_1, M_1) \rhd (N_1, N_2)}$$

$$ap(\lambda(x_1, \cdots, x_n).M.(N_1, \cdots, N_n)) \rhd M_{x_1,\cdots,x_n}[N_1, \cdots, N_n] \qquad (n \geq 1)$$

$$\frac{beval(A) = T}{if\ beval(A)\ then\ M\ else\ N \rhd M}$$

$$\frac{beval(A) = F}{if\ beval(A)\ then\ M\ else\ N \rhd N}$$

$$let\ (x_1, \cdots, x_n) = (T_1, \cdots, T_n)\ in\ M \rhd M_{x_1,\cdots,x_n}[T_1, \cdots, T_n]$$

$$\mu(z_1, \cdots, z_n).(M_1, \cdots, M_n) \rhd ((M_1)_{z_1,\cdots,z_n}[f_1, \cdots, f_n], \cdots, (M_n)_{z_1,\cdots,z_n}[f_1, \cdots, f_n])$$

where $\mu(z_1, \cdots, z_n).(M_1, \cdots, M_n) = (f_1. \cdots f_n)$
and $f_i = (M_n)_{z_1,\cdots,z_n}[f_1, \cdots, f_n]$ $(1 \leq i \leq n)$

Figure 1: Reduction Rules in λ^{qpc}

$$ap((M_1, \cdots, M_n), N) \equiv (ap(M_1, N), \cdots, ap(M_n, N))$$
$$\lambda X.(M_1, \cdots, M_n) \equiv (\lambda X.M_1, \cdots, \lambda X.N_n)$$
$$if\ A\ then\ (M_1, \cdots, M_n)\ else\ (N_1, \cdots, N_n)$$
$$\equiv (if\ A\ then\ M_1\ else\ N_1, \cdots, if\ A\ then\ M_n\ else\ N_n)$$
$$let\ X = T\ in\ (M_1, \cdots, M_n) \equiv (let\ X = T\ in\ M_1, \cdots, let\ X = T\ in\ M_n)$$

Figure 2: Term Equivalence Rules

The reduction rules of λ^{qpc} terms are given in Figure 1. The term equivalence relation, $\equiv$, is defined in Figure 2.

3 Types of QPC$_2$

3.1 Introducing *prop*, $type_1$, $type_2$ and $type_3$

The terms of the Calculus of Constructions (CoC) are classified into three levels: proof level, propositional type level and propositional scheme level [11, 14]. As **QPC$_2$** is also a higher order calculus, it has a similar classification of expressions. The essential difference is seen at the proof level. Our system is a type free formalism and the proofs are described with proof trees which correspond to the derivation trees in constructive type theories, so that there are no proof terms in our formalism. However, viewing realizability as a coding of proof trees, we can set a level called the *realizer level* which is an analogue to the proof level of CoC.

The realizers are regarded as programs in the first order type free constructive systems such as PX. On the other hand, a realizer is not always a program in $\mathbf{QPC}_2$. As will be explained in 5.1, the realizer of a universally quantified second order formula may contain lambda binding of predicate variables, and this binding is not a syntax of λ^{qpc}. The realizer with predicate variables will be called a *program scheme*. Therefore, the realizer level should have two sublevels: we set the *program level* as the lower sublevel and the *program scheme level* as the upper sublevel. As we aim to provide a program synthesis system for existing simply typed functional languages such as Standard ML, the realizability of our system is formulated as a typed realizability. The realizers at the program level are typed by a variant of simple type system for λ^{qpc}. On the other hand, the type system for the realizers at the program scheme level needs more types. Namely, the types for the predicate variables and the proposition variables. Consequently, we have a hierarchy of the type systems for realizers – one for the programs and another for the program schemes – and we introduce the constants $type_1$ and $type_3$. The types for programs are of type $type_1$ and the types for program schemes are of type $type_3$.

As a formula in $\mathbf{QPC}_2$ may contain second order universal quantification of the predicate variables and the proposition variables, the types of predicates and propositions are necessary to specify the domains of the quantifier. Therefore, we introduce the constant *prop*. Also, universal quantification over the simple types is also allowed, so that the constant $type_1$ may be used as the domain of the quantifier. The types used as the domains of the second order universal quantifier are of type $type_2$.

3.2 The type system

As explained in the previous subsection, the type system of $\mathbf{QPC}_2$ has three subsystems corresponding to $type\epsilon_1$, $type\epsilon_2$ and $type\epsilon_3$. We define them in the sequel.

(1) $type_1$ type system
The $type_1$ types are those of a simple type theory, and will be used as the type system for λ^{qpc} and for specifying the domains of the first order quantifiers in the formulas. Notice that there is no universal type quantifier because the type variables are always regarded as universally quantified outside the type expressions. Thus, this type system realizes the ML style polymorphism.

Def. 1: $type_1$ types

1) *nat*, *bool*, and **2** (primitive types) are $type_1$ types;
2) ϕ (empty type) is a $type_1$ type;
3) $\alpha, \beta, \cdots$ (type variables) are $type_1$ types;
4) If σ and τ are $type_1$ types, then $\sigma \times \tau$ (Cartesian product) and $\sigma \rightarrow \tau$ (arrow type) are $type_1$ types;
5) If σ is a $type_1$ type, then $L(\sigma)$ (type of lists over σ) is a $type_1$ type.

If an expression σ is a $type_1$ type, it is assumed that the typing relation $\sigma : type_1$ holds.

$$\frac{\sigma : type_1 \quad \tau : type_1 \quad M : \sigma \quad \sigma \equiv \tau}{M : \tau}$$

$$n : nat \quad (n = 0, 1, \cdots) \qquad \frac{\sigma : type_1 \quad \sigma \neq \phi}{any : \sigma}$$

$$T : bool \qquad F : bool \qquad left : 2 \qquad right : 2 \qquad () : \phi$$

$$\frac{\sigma : type_1}{nil : L(\sigma)} \qquad \frac{\sigma : type_1 \quad M : \sigma \quad N : L(\sigma)}{M :: N : L(\sigma)} \qquad \frac{\sigma : type_1 \quad M : L(\sigma) \quad N : L(\sigma)}{app(M, N) : L(\sigma)}$$

$$\frac{N : \sigma_1 \times \cdots \times \sigma_n}{tseq(k)(N) : \sigma_k \times \cdots \times \sigma_n} \quad (1 \leq k \leq n)$$

$$\frac{N : \sigma_1 \times \cdots \times \sigma_n}{ttseq(k, l)(N) : \sigma_k \times \cdots \times \sigma_{k+l-1}} \quad (1 \leq k \leq n, 1 \leq l \leq n - k + 1)$$

$$\frac{N : \sigma_1 \times \cdots \times \sigma_n}{proj(k)(N) : \sigma_k} \quad (1 \leq k \leq n)$$

$$\frac{\sigma : type_1 \quad M : \sigma \quad N : \sigma}{beval(M = N) : bool} \qquad \frac{M : nat \quad N : nat}{beval(M \ R \ N) : bool} \quad (R \equiv \leq, \geq, <, or >)$$

Figure 3: Typing Rules for λ^{qpc} Types (to be continued)

That is,

$$\frac{\sigma \ is \ a \ type_1 \ type}{\sigma : type_1}$$

The terms of λ^{qpc} are typed by the $type_1$ types and the typing rules are listed in Figure 3. Because λ^{qpc} contains the sequences of terms, the $type_1$ types should enjoy the following type equality rule to give the same typing to the equivalent terms:

$$\frac{\sigma : type_1 \quad \tau_i : type_1 \ (1 \leq i \leq n)}{\sigma \rightarrow (\tau_1 \times \cdots \times \tau_n) \equiv (\sigma \rightarrow \tau_1) \times \cdots \times (\sigma \rightarrow \tau_n)}$$

(2) $type_2$ type system
The $type_2$ types specify the domains of the second order universal quantifier ($\forall^2$). Notice that $\forall^2$ quantifies over both the predicates and the $type_1$ types, and the latter realizes a universal type quantification in the ML style polymorphism.

Def. 2: $type_2$ types
1) $type_1$ and *prop* (proposition type) are $type_2$ types;
2) If $\sigma : type_1$, then $\sigma \rightarrow prop$ (predicate type) is a $type_2$ type.

The typing rules for *prop* will be presented in the next section as the definition of formulas and predicates.

(3) $type_3$ type system
The $type_3$ type system includes the $type_1$ types and the additional types for program

$$\frac{\sigma : type_1 \quad M : L(\sigma) \quad M \neq nil}{hd(M) : \sigma} \qquad \frac{\sigma : type_1 \quad M : L(\sigma) \quad M \neq nil}{tl(M) : L(\sigma)}$$

$$\frac{\sigma : type_1 \quad \tau : type_1 \quad M : \sigma \quad N : \tau}{(M, N) : \sigma \times \tau}$$

$$\frac{[X : \sigma]}{\sigma : type_1 \quad \tau : type_1 \quad M : \tau}$$
$$\lambda X.M : \sigma \to \tau$$

$$\frac{\sigma : type_1 \quad \tau : type_1 \quad M : \sigma \to \tau \quad N : \sigma}{ap(M, N) : \tau}$$

$$\frac{\sigma : type_1 \quad \tau : type_1 \quad S : \sigma \quad T : \sigma \quad M : \tau \quad N : \tau}{if\ beval(S = T)\ then\ M\ else\ N : \tau}$$

$$\frac{\sigma : type_1 \quad S : nat \quad T : nat \quad M : \sigma \quad N : \sigma}{if\ beval(S\ R\ T)\ then\ M\ else\ N : \sigma} \quad (R = \leq, \geq, <\ or\ >)$$

$$\frac{[X : \sigma]}{\sigma : type_1 \quad T : \sigma \quad M : \tau}$$
$$let\ X = T\ in\ M : \tau$$

$$\frac{[Z : \sigma \to \tau]}{\sigma \to \tau : type_1 \quad T : \sigma \to \tau}$$
$$\mu Z.T : \sigma \to \tau$$

Figure 3 (end): Typing Rules for λ^{qpc} Types

schemes. An expression, M, of type $type_3$ is called a program scheme and if M is of $type_1$ it is called a program.

In order to define the $type_3$ types, the $type_1^+$ types are introduced. Before that we must introduce the $type(A)$ construct for a formula A. The realizer of a formula in $\mathbf{QPC}_2$ is of some type, and the type of the realizer can be determined mechanically only by the construction of the formula. $type()$ is a function which determines the types. If A is a first order formula, the realizers of A are programs so that the value of $type(A)$ is a $type_1$ type. But if A is a predicate variable or a proposition variable, the type of realizers of A is undefined and the evaluation of $type(A)$ is suspended until a first order predicate or a formula is substituted to A. Thus, the value of $type(A)$ is $type(A)$ itself. $type_1^+$ is an extension of the $type_1$ type system with these *undefined types*. The precise definition of the $type_1^+$ types is as follows.

Def. 3: $type_1^+$ types

1) nat, $bool$, and **2** are $type_1^+$ types;
2) ϕ is a $type_1^+$ type;
3) $\alpha, \beta, \cdots$ are $type_1^+$ types;
4) If σ and τ are $type_1^+$ types, then $\sigma \times \tau$ and $\sigma \to \tau$ are $type_1^+$ types;
5) If σ is a $type_1^+$ type, then $L(\sigma)$ is a $type_1^+$ type.
6) If P is a predicate variable, then $type(P)$ is a $type_1^+$ type.

The $type_3$ universe contains the $type_1^+$ types and the dependent product types over the domains of predicates and propositions.

Def. 4: $type_3$ types

1) If σ is a $type_1^+$ type, then σ is a $type_3$ type;
2) Let P be a predicate variable or a proposition variable. If τ is a $type_3$ type, then $(\Pi P \in \mathcal{K})\tau$ is a $type_3$ type, where $\mathcal{K}$ is a $type_2$ type such that $\mathcal{K} \neq type_1$.

The point is that, in the program extraction from the second order proofs in **QPC$_2$**, it is necessary to give the codes corresponding to the predicate variables and the second order universally quantified formulas. The former kind of codes will be typed using $type(P)$ and the latter kind codes will be typed by the dependent types in the form of $(\Pi P \in \sigma \rightarrow prop)\tau$ or $(\Pi P \in prop)\tau$.

3.3 Comparison with Martin-Löf universes U_1 and U_2

The constants $prop$ and $type_i$ $(i = 1, 2, 3)$ and Martin-Löf's universes U_1 and U_2 have some similarity, but the simple cumulative hierarchy of U_1 and U_2 is destroyed in our calculus which is mainly because of the restriction of ITT and the distinction between types and formulas. The hierarchy of our type system can be informally described as follows:

$$type_1 \in type_2 \quad (1)$$
$$prop \in type_2 \quad (2)$$
$$type_1 \subset type_3 \quad (3)$$

This hierarchy and the characterization of each constant make clear the relation with the universes U_1 and U_2. First of all, $type_1$ is an analogue of U_1. They are both universes for small types. Unlike Martin-Löf's system, types and formulas are not identified in **QPC$_2$**, so that we also introduced $prop$ universe as another analogue of U_1. Secondly, $type_2$ and $type_3$ have similar natures to U_2 universe. U_2 contains U_1, and this corresponds to (1) and (2). U_2 also contains the elements of U_1, and this corresponds to (3). One would wonder why our type universes do not have a clear cumulative hierarchy like U_1 and U_2 such as

$$type_1 \in type_3$$
$$type_1 \subset type_2$$
$$prop \subset type_2$$
$$prop \in type_3$$
$$prop \subset type_3.$$

The first relation means that a program scheme may contain the type variables of type $type_1$. But the type variables are not the syntax of our programming language and they are stripped from the proofs in the program extraction. Therefore, this relation is redundant in our calculus. The second relation may be allowed if we characterize the $type_2$ types to be the domains of all the quantifiers. But we restrict $type_2$ universe to the collection of

the domains of the second order universal quantifier, so that this relation does not hold. Notice that we do not use $type_2$ constant extensively in our calculus. It is only used for describing the definition of the second order formulas as will be shown in the next section. Thus, the restriction of $type_2$ universe is not essential. The third relation does not hold in our calculus because the formulas of type *prop* will never be the domains of the second order quantifier. The fourth relation does not hold because a program scheme is not a formula or a predicate. The final relation does not hold because a realizer of a formula A is not typed by the formula but by $type(A)$. Consequently, the cumulative hierarchy of Martin-Löf's universes is destroyed in $\mathbf{QPC}_2$ because of the distinction of types and formulas. Of course, $type_2$ and $type_3$ universes are much smaller than U_2 universe, and they are disjoint as well as $type_1$ and *prop*.

Another difference from Martin-Löf's ITT is that, as will be given in 4.3, $\mathbf{QPC}_2$ has elimination rules of the second order universal quantifier which enable the quantification over predicates. The original version of ITT [10] does not have the elimination rules for the dependent type symbols in the universes while it can be naturally extended to the theory with the rules.

4 Logic of $\mathbf{QPC}_2$

The second order formulas of $\mathbf{QPC}_2$ are restricted in the sense that the second order universal quantifier never occurs inside a formula. This restriction allows to make the order of the second order proof normalization drastically simple. The practical meaning of this nice property is as follows. First one proves a general specification using the predicate variables as parameters. When one need to develop a particular program instantiating the parameters of a general specification, one proves the first order formulas which will be substituted to the parameters and describes the substitution procedure as a few steps of proof procedure. Then the system takes care of the rest. The second order proof normalization is performed automatically removing the predicate variables in the proof, and generates a λ^{qpc} program.

The normalization procedure can also be carried out at the term level like other constructive calculi based on formulas-as-type notion. To describe the codes extracted from the proofs of general specifications, we introduce the notion of *program schemes* which resemble the second order lambda terms and the reduction of a program scheme corresponds to the normalization procedure of a proof of a general specification.

4.1 Formulas

A formula of $\mathbf{QPC}_2$ is either a first order formula, a second order formula without the second order quantifier or a second order formula with the second order universal quantifier only at the head part of the formula. This can be understood as an analogue of the type

expressions in ML style polymorphism in which the universal type quantifier does not occur inside a type expression.

The formulas without the second order universal quantifier are at the *prop* level, and they can be formally defined by a set of formation rules of expression of type *prop*. We do not adopt the type theoretic formulation of the formulas here but follow the conventional style of mathematical logic in defining them. We call the formulas at *prop* level the *class 1 formulas*, and the formulas with second order universal quantifiers are called the *class 2 formulas*. Notice that 'class 1' and 'class 2' are not part of the syntax of our calculus.

In the following, P, Q, $\cdots$ denote the predicate variables. Each predicate variable is assigned a natural number called *arity* which means the number of parameters. A predicate variable with arity 0 is called a proposition variable, so that we will regard the proposition variables as a special case of the predicate variables.

Def. 5: Class 1 formula

1) If M and N are terms, then $M = N$, $M < N$, $M \leq N$, and $\bot$ are class 1 formulas (atomic formulas);
2) If P is a n-ary predicate variable and $M_1 \cdots M_n$ are terms, then $P(M_1, \cdots, M_n)$ is a class 1 formula;
3) If A and B are class 1 formulas and σ is a $type_1$ type, then $A \wedge B$, $A \vee B$, $A \supset B$, $\forall x \in \sigma.A(x)$, and $\exists x \in \sigma.A(x)$ are class 1 formulas.

Negation of a formula, A, is defined as $\neg A \stackrel{\text{def}}{=} A \supset \bot$.

Def. 6: Class 2 formula

1) If A is a class 1 formula, then A is a class 2 formula;
2) If A is a class 1 formula, then $\forall^2 X_1 \in \sigma_1. \cdots \forall^2 X_n \in \sigma_n.A$ is a class 2 formula where σ_is are $type_2$ types.

Notice that, in the second clause of **Def. 6**, if $\sigma_i = type_1$ then X_i is a type variable, otherwise, X_i is a predicate variable. $\forall^2$ quantifies both the predicate variables and the type variables, but they are clearly separated by the types.

Def. 7: Predicate
If A is a class 1 formula which contains free occurrences of individual variables $x_1, \cdots, x_n$, then an expression, $\lambda(x_1, \cdots, x_n).A$ is called a *predicate* or an *abstract*.

The class 1 formulas and the predicates are typed by the $type_2$ types, and the typing rules are as follows:

$$\frac{A \text{ is a } class_1 \text{ formula}}{A : prop}$$

$$\frac{[\overline{x} : \sigma]}{\dfrac{A : prop \quad \sigma : type_1}{\lambda \overline{x}.A : \sigma \rightarrow prop}} \qquad \frac{\sigma : type_1 \quad \lambda \overline{x}.A : \sigma \rightarrow prop \quad T : \sigma}{A_{\overline{x}}[T] : prop}$$

A class 2 formula in the form of $\forall^2 X.A$ corresponds to a type in the second universe of ITT, but it is not in the $type_2$ or the $type_3$ universes. There is no need of introducing the typing rules for all of the class 2 formulas or predicates with the predicate variables because second order universal quantification is restricted.

4.2 Program Scheme

The program extraction using realizability is well established for the first order constructive logics, and extraction of λ^{qpc} terms can be carried out using the standard method in the first order fragment of $\mathbf{QPC}_2$. However, for the full calculus of $\mathbf{QPC}_2$, handling of formulas with the predicate variables and quantification over them is rather problematic. From a purely logical point of view, a forgetful interpretation of the second order quantifier will work well. But since we aim to provide a system with parameterized specifications in which predicate variables are used as the parameters, it is necessary that codes with the parameters are extracted from the second order proofs and programs are obtained by substituting suitable codes to the parameters.

One solution is to view the occurrences of the predicate variables in a proof as unspecified first order predicates. The program extractor generates λ^{qpc} codes from well specified part of the given proofs and, from the unspecified part of the proof it generates the expressions representing the extraction procedure which will be postponed until the predicate variables are well specified. This is the basic idea behind the notion of *program schemes*.

A program scheme is roughly a mixture of the terms and the predicate variables. The program schemes are used to describe the codes extracted from the proofs in $\mathbf{QPC}_2$. If a program scheme does not contain any predicate variables, it is a program. Basically we use a lambda binding of predicate variables which resembles the lambda binding of type variables in the higher order type theories. But program schemes are more complex because the realizers of the first order formulas are sequences of terms in $\mathbf{QPC}_2$ and we need more constructs, namely Rv-schemes and dummy code schemes.

As will be explained in section 5, the program extraction from the proofs in the first order fragment of $\mathbf{QPC}_2$ uses the *realizing variable sequences*, $Rv(A)$, and the *dummy code sequences*, $any[len(A)]$ where A is a formula. The realizing variable sequences are the codes extracted from the discharged hypotheses, and the dummy code sequences are the codes extracted from the proofs in $(\bot E)$ (absurdity elimination rule) and the disjunction introduction rules. The length of a realizing variable sequence or a dummy code sequence represents the amount of the extracted code, and it is calculated by a meta level function *len* according to the syntactical construction of the proved formula. So that the length is referred to as the *length of a formula*. The notion of length is extensively used in the extended projection method. For the full calculus of $\mathbf{QPC}_2$, predicate variables must be

taken into account, and the length of a predicate variable (applied to some terms) cannot be calculated. Therefore, the Rv-scheme and the dummy code scheme are introduced as the devices to delay the calculation of the length until the predicate variables are instantiated. We denote $L(P(M_1, \cdots, M_n))$ the undefined length of a formula with a predicate variable P.

Def. 8: Program scheme

1) If P is a predicate variable and M_i $(1 \leq i \leq n)$ is a term, then $any[L(P(M_1, \cdots, M_n))]$ (dummy code scheme) and $RV(P(M_1, \cdots, M_n))$ (Rv-scheme) are program schemes;
2) If P is a predicate variable and T is a program scheme, then $\Lambda P.T$ (Λ-term) is a program scheme;
3) The terms formed by regarding a dummy code sequence as a term and a Rv-scheme as a variable are program schemes;
4) If T is a program scheme and if Q is a predicate variable or a predicate, then $ap(T, Q)$ is a program scheme.

The term, $any[n]$, can be naturally extended to the case in which n is replaced by $m + L(P(M_1, \cdots, M_n))$ or $L(P(M_1, \cdots, M_n)) + m$ for a natural number m:

$$any[m + L(P(M_1, \cdots, M_n))] \equiv (any[m], any[L(P(M_1, \cdots, M_n))])$$

$$any[L(P(M_1, \cdots, M_n)) + m] \equiv (any[L(P(M_1, \cdots, M_n))], any[m])$$

The predicate variables in an extracted code are bound by Λ, and instantiation of the variables is performed as a β-reduction

$$ap(\Lambda P.T, S) \rhd T_P[S]$$

followed by the calculation of Rv-schemes and the dummy code schemes with the following reduction rules:

$$RV((\lambda X.p)(M)) \rhd Rv(p_X[M])$$

$$any[L((\lambda X.p)(M))] \rhd any[len[p_X[M]]]$$

where $\lambda X.p$ is an abstract, $(\lambda X.p)(M)$ is obtained by substitution of $\lambda X.p$ into the predicate variable P in $P(M)$, and *len* is an extension of the notion of length of a formula in the first order fragment of the system. The precise definition of *len* is as follows.

Def. 9: *len*

Let A be a class 1 formula, then

1) $len(A) = 0$ if A is atomic;
2) $len(P(M_1, \cdots, M_n)) = L(P(M_1, \cdots, M_n))$ if P is a predicate variable;
3) $len(A \wedge B) = len(A) + len(B)$;
4) $len(A \supset B) = len(B)$;
5) $len(\forall x \in \sigma.A) = len(A)$;
6) $len(A \vee B) = 1 + len(A) + len(B)$;

7) $len(\exists x \in \sigma.A) = 1 + len(A)$.

Before presenting the typing rules for program schemes, it is necessary to give the precise definition of the $type(\cdots)$ construct introduced in 3.2.

Def. 10: $type(A)$:
Let A be a class 2 formula or a predicate variable. $type(A)$, which reads "the type assigned to A", is defined by the following equations:

1) $type(A \wedge B) = type(A) \times type(B)$;
2) $type(A \vee B) = \mathbf{2} \times type(A) \times type(B)$;
3) $type(A \supset B) = type(A) \rightarrow type(B)$;
4) $type(\forall x \in \sigma.A) = \sigma \rightarrow type(A)$;
5) $type(\exists x \in \sigma.A) = \sigma \times type(A)$;
6) $type(A) = \phi$; when A is atomic;
7) $type(P) = type(P(M_1, \cdots, M_n))$ if P is a predicate variable and M_is are any terms;
8) $type(\forall^2 \alpha \in type_1.A) = type(A)$;
9) $type(\forall^2 P : \mathcal{K}.A) = (\Pi P \in \mathcal{K})type(A)$ where $\mathcal{K}$ is a $type_2$ type such that $\mathcal{K} \neq type_1$.

Proposition 1 *Let A be a class 2 formula, M and N be arbitrary terms, P be a predicate variable, α be a type variable, σ be a $type_1$ type and S be an abstract of the same type as that of P. Then,*
(1) $type(A(M)) = type(A(N))$;
(2) $type(A_P[S]) = type(A)_P[S]$;
(3) $type(A)_\alpha[\sigma] = type(A_\alpha[\sigma])$;
(4) $type(A)$ *is a* $type_3$ *type.*

The typing rules for the program schemes are as follows. Recall that in the first clause of the definition of the $type_3$ types, the $type_1^+$ types are $type_3$ types. The typing rules with regard to the $type_1^+$ types are obtained by the rules given in Figure 3 by replacing '$type_1$' by '$type_1^+$'. The obtained rules form a part of the typing rules for the program schemes without Λ-abstraction. In addition to them, the following rules are necessary.

$$any[len(A)] : type(A)$$

$$Rv(A) : type(A)$$

where A is a class 1 formula. The precise definition of Rv will be given in section 5.
Let $\mathcal{K}$ be either $\sigma \rightarrow prop$ or $prop$ in the $type_2$ universe.

$$\frac{P : \mathcal{K}}{RV(P(\overline{M})) : type(P(\overline{M}))} \qquad \frac{P : \mathcal{K}}{any[L(P(\overline{M}))] : type(P(\overline{M}))}$$

$$\frac{\begin{array}{c}[P : \mathcal{K}] \\ T : \tau\end{array}}{\Lambda P.T : (\Pi P \in \mathcal{K})\tau} \qquad \frac{\Lambda P.T : (\Pi P \in \mathcal{K})\tau \quad S : \mathcal{K}}{ap(\Lambda P.T, S) : \tau_P[S]}$$

Example: Let $P : nat \rightarrow prop$, then $\Lambda P.RV(P(0)) : (\Pi P \in nat \rightarrow prop).type(P(0))$. Now let $S \stackrel{\text{def}}{=} \lambda x.\exists y \in nat.x = y : nat \rightarrow prop$. Then $ap(\Lambda P.RV(P(0)), S) : type(P(0))_P[S]$ where $type(P(0))_P[S] = type(\exists y \in nat.0 = y) = nat$. Also, $ap(\Lambda P.RV(P(0)), S) \rhd RV(S(0)) \rhd Rv(\exists y \in nat.0 = y)$ and $Rv(\exists y \in nat.0 = y) : nat$.

4.3 Logical Rules

The rules for class 1 and class 2 formulas are given here.

(1) Rules for class 1 formulas

The rules for class 1 formulas are those of first order intuitionistic natural deduction with equality and induction on natural numbers and finite lists. The induction rules in $\mathbf{QPC_2}$ are slightly general ones:

$$\frac{[x = 0 \vee (x \neq 0 \wedge A(pred(x)))] \quad A(x)}{\forall x \in nat.A(x)}(nat\text{-}ind) \qquad \frac{[x = nil \vee (x \neq nil \wedge A(tl(x)))] \quad A(x)}{\forall x \in L(\sigma).A(x)}(L(\sigma)\text{-}in$$

Besides the standard equality rules in first order logic, the followings are needed to commute with the $type_1$ types and λ^{qpc}.

$$\frac{\sigma : type_1 \quad M : \sigma}{M = M} \qquad \frac{M \equiv N}{M = N} \qquad \frac{M \rhd N}{M = N}$$

(2) Rules for class 2 formulas

- $\forall^2 I$ rule:

$$\frac{\begin{matrix}[X : \mathcal{K}] \\ A\end{matrix}}{\forall^2 X \in \mathcal{K}.A}(\forall^2 I)$$

where $\mathcal{K}$ is a $type_2$ type.

- $\forall^2 E$ rule:

$$\frac{S : \mathcal{K} \quad \forall^2 X \in \mathcal{K}.A}{A_X[S]}(\forall^2 E)_1$$

where $\mathcal{K}$ is either $type_1$ or $prop$.

$$\frac{\lambda\overline{x}.p : \sigma \rightarrow prop \quad \forall^2 P \in \sigma \rightarrow prop.A}{Conv(A_P[\lambda\overline{x}.p])}(\forall^2 E)_2$$

$Conv(A_P[\lambda\overline{x}.p])$, also denoted just $A_P[\lambda\overline{x}.p]$, is obtained by performing a β-reduction

$$(\lambda\overline{x}.p)(M) \stackrel{\text{def}}{=} ap(\lambda\overline{x}.p, M) \rhd p_{\overline{x}}[M]$$

in $A_P[\lambda\overline{x}.p]$.

4.4 Some Properties

The following properties are easily proved.

Proposition 2 *Assume that A is a class 2 formula in the form of $\forall^2 X \in \sigma.F$. Then, if A is proved in* $\mathbf{QPC}_2$, *the last rule used in the proof is either $(\forall^2 I)$, $(\forall^2 E)_1$ or $(\forall^2 E)_2$*

This is clear from the restriction of the class 2 formulas and from the fact that any first order rules are only for the class 1 formulas.

Proposition 3 *Assume a class 2 formula $\forall^2 X_1 \in \sigma_1. \cdots \forall^2 X_n \in \sigma_n.A$ where A is a class 1 formula. If the formula is provable in* $\mathbf{QPC}_2$, *then there exists a proof in* $\mathbf{QPC}_2$ *in which last n rules are all $(\forall^2 I)$, and the rule is not used elsewhere.*

This can be proved by using the proof normalization on the second order universal quantifier.

Corollary:
If a class 1 formula, A, is proved in $\mathbf{QPC}_2$, *there is a proof of A which does not use $(\forall^2 I)$, $(\forall^2 E)_1$, or $(\forall^2 E)_2$*

This is proved by a straightforward application of the proof normalization and this indicates the practically nice property of $\mathbf{QPC}_2$ explained at the beginning of this section.

Prawitz coding of logical connectives and quantifiers is impossible in $\mathbf{QPC}_2$. For example, a disjunction $\oplus$ may be defined as

$$A \oplus B \equiv \forall^2 P \in prop.(A \supset B) \supset (B \supset P) \supset P$$

If A is proved, the following deduction can be simulated as usual,

$$\frac{\begin{array}{c}\Sigma \\ A\end{array}}{A \oplus B}$$

but it is impossible to simulate

$$\frac{\dfrac{[A]}{A \oplus B}}{A \supset A \oplus B}(\supset I)$$

because the $(\supset I)$ rule is only allowed for class 1 formulas but $A \oplus B$ is a class 2 formula. Therefore, although $\mathbf{QPC}_2$ is a second order logic with primitive logical connectives and the first order existential quantifier, they are not redundant.

5 Program Extraction

The precise definition of realizability interpretation of $\mathbf{QPC}_2$ will be given in this section. A variant of typed q-realizability interpretation is given to $\mathbf{QPC}_2$, and the proof of

soundness theorem of the interpretation gives the program extraction algorithm. The interpretation of logical connectives and the first order quantifiers is rather standard. For the second order quantifier, the realizability gives a Kreisel-Troelstra style forgetful interpretation to the quantification over $type_1$ and an interpretation similar to second order typed lambda calculus is given to the quantification over other $type_2$ types. Λ-terms are extracted from the second order proofs by the latter interpretation.

5.1 qpc-realizability

Assume that A is a class 2 formula and that $\overline{z} = (z_1, \cdots, z_n)$ is a sequence of fresh variables of suitable length. Then, $\overline{z}$ **qpc** A, which reads "$\overline{z}$ realizes A", is called a **qpc**-realizability relation, or **qpc**-realizability for short. $\overline{z}$ is called a sequence of realizing variables, or a realizing variable sequence, of A. The realizability relations are defined as follows:

Def. 11: qpc-realizability

1) If A is atomic, then $()$ **qpc** $A \stackrel{\text{def}}{=} A$;
2) If P is a predicate variable, then $\overline{a}$ **qpc** $P(M_1, \cdots, M_n) \stackrel{\text{def}}{=} \overline{a} = RV(P(M_1, \cdots, M_n)) \wedge P(M_1, \cdots, M_n)$;
3) $\overline{a}$ **qpc** $A \supset B \stackrel{\text{def}}{=} (A \supset B) \wedge \forall b \in type(A).(b$ **qpc** $A \supset ap(\overline{a}, b)$ **qpc** $B)$;
4) $(a, \overline{b})$ **qpc** $\exists x \in \sigma.A \stackrel{\text{def}}{=} a : \sigma \wedge A_x[a] \wedge \overline{b}$ **qpc** $A_x[a]$;
5) $\overline{a}$ **qpc** $\forall x \in \sigma.A \stackrel{\text{def}}{=} \forall x \in \sigma.(ap(\overline{a}, x)$ **qpc** $A)$;
6) $(z, \overline{a}, \overline{b})$ **qpc** $A \vee B \stackrel{\text{def}}{=} (z = left \wedge A \wedge \overline{a}$ **qpc** $A) \vee (z = right \wedge B \wedge \overline{b}$ **qpc** $B)$;
7) $(\overline{a}, \overline{b})$ **qpc** $A \wedge B \stackrel{\text{def}}{=} \overline{a}$ **qpc** $A \wedge \overline{b}$ **qpc** B;
8) $\overline{a}$ **qpc** $\forall^2 P \in \mathcal{K}.A \stackrel{\text{def}}{=} \forall^2 P \in \mathcal{K}.(ap(\overline{a}, P)$ **qpc** $A)$ where $\mathcal{K}$ is a $type_2$ type other than $type_1$;
9) $\overline{a}$ **qpc** $\forall^2 \alpha \in type_1.A \stackrel{\text{def}}{=} \forall^2 \alpha \in type_1.(\overline{a}$ **qpc** $A)$.

Notice that there are two kinds of interpretation of $\forall^2$. The clause 9) is the same as Kreisel-Troelstra realizability [9]. The intention of clauses 8) and 9) is that predicates that have computational meaning should be preserved in the program extraction while the type information should be removed.

Here we should explain the role of the empty sequence, $()$. The empty sequence is used as the empty code in the program extraction which can be seen in the first clause in Def. 11. In the standard q-realizability interpretation [22], any computational meaning (realizer) can be given to an atomic formula. So that we can give, for example, axiom names to atomic formulas as their realizers. However, as the atomic formulas in our calculus are (in)equalities of terms or $\perp$ (abort) which can be directly executed on computers, no additional computational meaning should be extracted from proofs. Therefore, the realizer of an atomic formula may be empty and $()$, which is of type the empty type, ϕ, is the code representing the emptyness. This idea was first introduced as **px**-realizability in

the PX system [8] where the nil list term, nil, is used for the realizer of atomic formulas in PX.

From the definition of qpc-realizability, a sequence of realizing variables can be determined as follows by the structure of the given formula and the equivalence rules of terms:

Def. 12: $Rv(A)$ (sequence of realizing variables)

1) $Rv(A) = ()$ if A is atomic;
2) $Rv(P(M_1, \cdots, M_n)) = RV(P(M_1, \cdots, M_n))$ if P is a predicate variable;
3) $Rv(A \wedge B) = (Rv(A), Rv(B))$;
4) $Rv(A \supset B) = Rv(B)$;
5) $Rv(\forall x \in \sigma.A) = Rv(A)$;
6) $Rv(A \vee B) = (z, Rv(A), Rv(B))$ (z is a new variable);
7) $Rv(\exists x \in \sigma.A) = (z, Rv(A))$ (z is a new variable);
8) $Rv(\forall^2 X \in \mathcal{K}.A) = Rv(A)$ where $\mathcal{K}$ is a $type_2$ type.

Notice that $RV(P(M_1, \cdots, M_n))$ is regarded as a variable.

Def. 13: $\mathbf{QPC}_2^+$
$\mathbf{QPC}_2^+$ is a trivial extension of $\mathbf{QPC}_2$ by adding all the realizability relations as formulas. Precisely, z qpc A is a class n formula in $\mathbf{QPC}_2^+$ when A is class n formula in $\mathbf{QPC}_2$ ($n = 1, 2$).

Def. 14: Let A be a formula and M be a program scheme, then
M qpc $A \stackrel{\text{def}}{=} (Rv(A)$ qpc $A)_{Rv(A)}[M]$

The following proposition, which is necessary to prove the soundness of qpc-realizability, can be proved by induction on the construction of A.

Proposition 4 *Let M and N be program schemes and A be a formula, then* $(M$ qpc $A)_x[N] \Leftrightarrow M_x[N]$ qpc $A_x[N]$ *holds in* $\mathbf{QPC}_2^+$.

5.2 Properties of qpc-realizability

Proposition 5 *Let A be a class 2 formula. If A is realizable, i.e, there is a program scheme M such that M* qpc *A, then A is provable in* $\mathbf{QPC}_2$.

Proof: By induction on the construction of A and the definition of qpc-realizability. ∎

In the following theorem, $FV(M)$ denotes the set of free variables of an expression.

Theorem Soundness of qpc-realizability
Assume that A is a class 2 formula in $\mathbf{QPC}_2$. *If A is proved in* $\mathbf{QPC}_2$, *then*

(1) there is a program scheme M such that M **qpc** *A can be proved in* $\mathbf{QPC}_2^+$;
(2) $M : type(A)$;
(3) $FV(M) \subset FV(A)$.

The proof is given in **Appendix**.

Corollary:
Assume that A is a class 1 formula which does not contain any predicate variables. If A is proved in $\mathbf{QPC}_2$, *then there is a term M which realizes A. If A is a closed formula, then M does not contain free variables.*

The corollary means that the program extraction in $\mathbf{QPC}_2$ is an extension of that in the first order fragment. Moreover, the type $type(A)$ of the term M is a $type_1$ type and M is actually a program.

Notice that the extracted programs do not have the termination property. Because $\mathbf{QPC}_2$ only uses, as the induction rules, the mathematical induction and the structural induction on lists, the extracted programs seem to have the property. But μ-terms are allowed as the typed terms in λ^{qpc}, nonterminating terms can be introduced in the proof procedure in the application of $(\exists I)$ and $(\forall E)$ rules (as the terms M and N in the following figures):

$$\frac{\begin{matrix}\Sigma_0 & \Sigma_1 \\ M : \sigma & A(M)\end{matrix}}{\exists x \in \sigma.A(x)}(\exists I) \qquad \frac{\begin{matrix}\Sigma_0 & \Sigma_1 \\ N : \sigma & \forall x \in \sigma.A(x)\end{matrix}}{A(N)}(\forall E)$$

For example, consider the formula $\forall x : nat.\exists y : L(nat).\ x = hd(y)$. This formula can be proved as follows:

$$\frac{\dfrac{\dfrac{\begin{matrix}\Sigma_1 \\ \mu z.\lambda n.n :: ap(z, n+1) : nat \to L(nat)\end{matrix} \quad [x : nat]}{ap(\mu z.\lambda n.n :: ap(z, n+1), x) : L(nat)} \quad \begin{matrix}\Sigma_2 \\ x = hd(ap(\mu z.\lambda n.n :: ap(z, n+1), x))\end{matrix}}{\exists y : L(nat).\ x = hd(y)}(\exists I)}{\forall x : nat.\exists y : L(nat).\ x = hd(y)}(\forall I)$$

where Σ_1 is

$$\dfrac{\dfrac{\dfrac{[n : nat] \quad \dfrac{[z : nat \to L(nat)][n : nat]}{ap(z, n+1) : L(nat)}}{n :: ap(z, n+1) : L(nat)}}{\lambda n.n :: ap(z, n+1) : nat \to L(nat)}}{\mu z.\lambda n.n :: ap(z, n+1) : nat \to L(nat)}$$

and Σ_2 is

$$\frac{x = hd(x :: ap(MU, x)) \quad ap(MU, x) = x :: ap(MU, x)}{x = hd(ap(\mu z.\lambda n.n :: ap(z, n+1), x))}(= E)$$

where $MU \stackrel{\text{def}}{=} \mu z.\lambda n.n :: ap(z, n+1)$

The code extracted from this proof is $Code \stackrel{\text{def}}{=} \lambda x.ap(\mu z.\lambda n.n :: ap(z, n+1), x)$ and $ap(Code, 0)$, for example, does not terminate.

In order to assure the termination property of the extracted programs, one can modify our calculus to a logic of partial terms, such as PX, in which $(\exists I)$ and $(\forall E)$ rules are restricted:

$$\frac{Terminates(M) \quad M : \sigma \quad A(M)}{\exists x \in \sigma.A(x)}(\exists I) \qquad \frac{Terminates(N) \quad N : \sigma \quad \forall x \in \sigma.A(x)}{A(N)}(\forall E)$$

where the predicate $Terminates(\ )$ means that the evaluation of the term terminates and denotes a value. The easiest, but too restricted, definition of the predicate is $Terminates(M) =$ "M *is not a* $\mu -$ *term*" and in this case we understand that recursive call programs should always be defined with the induction rules in our calculus.

5.3 The Program Extractor: Ext

The proof of the soundness theorem in the previous subsection can be formalized as a program extraction procedure, Ext, in a straightforward way as in [17, 19]. It suffices, here, to show the cases of the proofs in the second order rules and an induction.

$$Ext\left(\frac{\begin{matrix}[X : \mathcal{K}] \\ \Sigma \\ B\end{matrix}}{\forall^2 X \in \mathcal{K}.B}(\forall^2 I)\right) = \begin{cases} Ext\begin{pmatrix}\Sigma \\ B\end{pmatrix} & \text{if } \mathcal{K} = type_1 \\ \Lambda X.\ Ext\begin{pmatrix}\Sigma \\ B\end{pmatrix} & \text{if } \mathcal{K} \text{ is other } type_2 \text{ type} \end{cases}$$

$$Ext\left(\frac{\begin{matrix}\Sigma_0 & \Sigma_1 \\ S : \mathcal{K} & \forall^2 X \in \mathcal{K}.A\end{matrix}}{A_X[S]}(\forall^2 E)_i\right) = \begin{cases} Ext\begin{pmatrix}\Sigma_1 \\ \forall^2 X \in \mathcal{K}.A\end{pmatrix} & \text{if } \mathcal{K} = type_1 \text{ and } i = 1 \\ ap\left(Ext\begin{pmatrix}\Sigma_1 \\ \forall^2 X \in \mathcal{K}.B\end{pmatrix}, S\right) & \text{if } \mathcal{K} \text{ is other } type_2 \text{ type and } i = 1 \text{ or } 2 \end{cases}$$

$$Ext\left(\frac{\begin{matrix}[x = nil \vee (x \neq nil \wedge B(tl(x)))] \\ \Sigma \\ B(x)\end{matrix}}{\forall x \in L(\sigma).B(x)}(L(\sigma)\text{-}ind)\right)$$

$$= \mu\overline{z}.\lambda x.Ext\begin{pmatrix}\Sigma \\ B(x)\end{pmatrix}_{Rv(H)}[if\ x = nil\ then\ left\ else\ right, ap(\overline{z}, tl(x))]$$

where $H \stackrel{\text{def}}{=} x = nil \vee (x \neq nil \wedge B(tl(x)))$.
By assuming $(\Sigma/B(x))$ proved by the $(\vee E)$ rule with regard to the disjunction H, Ext for $(L(\sigma)\text{-}ind)$ is specialized as follows.

$$Ext\left(\frac{\begin{matrix}[x \neq nil \wedge B(tl(x))] \\ \Sigma_1 \quad \Sigma_2 \\ B(nil) \quad B(x)\end{matrix}}{\forall x \in L(\sigma).B(x)}(L(\sigma)\text{-}ind)\right)$$

$$= \mu\overline{z}.\lambda x.\ if\ (if\ x = nil\ then\ left\ else\ right) = left$$
$$then\ Ext\begin{pmatrix}\Sigma_1\\ B(nil)\end{pmatrix}_{Rv(H)}[if\ x = nil\ then\ left\ else\ right, ap(\overline{z}, tl(x))]$$
$$else\ Ext\begin{pmatrix}\Sigma_2\\ B(x)\end{pmatrix}_{Rv(H)}[if\ x = nil\ then\ left\ else\ right, ap(\overline{z}, tl(x))]$$

$if\ x = nil\ then\ left\ else\ right = left$ is equivalent to $x = nil$. $Rv(H)$ does not occur in the $Ext(\Sigma_1/B(nil))$ part. $Rv(H) = (z, Rv(tl(x)))$ for a fresh variable z, and z does not occur in the $Ext(\Sigma_2/B(x))$ part. Therefore, the extracted code is equal to

$$\mu\overline{z}.\lambda x.if\ x = nil\ then\ Ext\begin{pmatrix}\Sigma_1\\ B(nil)\end{pmatrix}\ else\ Ext\begin{pmatrix}\Sigma_2\\ B(x)\end{pmatrix}_{Rv(B(tl(x)))}[ap(\overline{z}, tl(x))]$$

According to the properties of the $\mathbf{QPC_2}$ proofs presented in 4.4, it is easy to observe that the program scheme generated from a $\mathbf{QPC_2}$ proof by Ext and the second order proof normalization is a λ^{qpc} term or a program scheme in the form of $\Lambda P_1.\cdots\Lambda P_n.M$ where M is a program scheme which may contain the predicate variables, P_i $(i = 1, 2, \cdots n)$ and does not contain Λ binding.

Note: In the following, particularly in the examples given in section 7, the realizing variable sequence, $Rv(B(tl(x)))$, will be used instead of the μ-bound parameter $\overline{z}$ in the extracted codes. This is a trivial trick to avoid introducing a new program scheme representing a sequence of fresh variables which has the same length as the conclusion of a given induction proof.

5.4 Optimization

Several optimization techniques have been developed mainly for the first order constructive calculi such as the proof normalization method for partial evaluation, the modified $\vee$ code method [17] and the pruning rule [6] for eliminating redundancy in decision procedures, and various techniques to remove computationally irrelevant codes [12, 1, 8, 18, 19]. All of them can be used for $\mathbf{QPC_2}$ because it is an extension of a first order calculus. Among the last technique, the extended projection method [19] will be used extensively in the following examples. This technique is to remove redundancy in the extracted codes. For example, from a proof of $\exists x \in \sigma.A(x)$, qpc-realizability extracts a code in the form of (t, s) in which t is a term such that $A(t)$ holds, and s is a term extracted from the subproof of $A(t)$. The code s is often redundant. In the extended projection method, position numbers are assigned to the occurrences of the symbols $\exists$ and $\vee$ in a formula and computationally irrelevant symbols are specified with the numbers. The position numbers are defined with the length of the formula. As explained in 4.2, if the predicate variables occur in a formula, the length of the formula is undefined and the position numbers cannot be determined. Consequently, we use the extended projection method when all the predicate variables are instantiated and $\mathbf{QPC_2}$ is designed to be compatible with this method.

6 Specifying Parameterized Programs

Two examples, a map-function and a general sorting program, are presented to demonstrate the programming technique in $\mathbf{QPC}_2$.

6.1 Map Function

Two kinds of specification of map-functions are possible.

(1) Specification using a function variable
We use a function variable, f, for the input function of the map-function. The type of the input function is parameterized.

$$\begin{aligned}&\forall^2\alpha \in type_1.\forall^2\beta \in type_1.\forall f \in \alpha \to \beta.\\ &\quad \forall x \in L(\alpha).\exists y \in L(\beta)\\ &\qquad length(x) = length(y)\\ &\qquad \wedge(\forall i \in nat.1 \leq i \leq length(x) \supset f(elem(i,x)) = elem(i,y))\end{aligned}$$

where $length(x)$ and $elem(i,x)$ are functions which calculates the length of x and the ith element of x. They are definable in λ^{qpc}.
The specification can be proved by $(\forall^2 I)$, $(\forall I)$, and $(L(\alpha)\text{-}ind)$. The extracted code will be

$$\lambda f.\mu z.\lambda x.if\ x = nil\ then\ nil\ else\ ap(f, hd(x)) :: ap(z, tl(x)).$$

(2) Specification using a predicate variable
The application of the above map-function to a function is carried out at the program level. If one needs to carry out the application at the proof description level or needs to define a general scheme of recursive call programs, one can use the predicate variables as the parameters.
The input function of the map function can be parameterized as $\forall x.\exists y.P(x,y)$ where P is a predicate variable.

$$\begin{aligned}&\forall^2\alpha \in type_1.\forall^2\beta \in type_1.\forall^2 P \in \alpha \times \beta \to prop.\\ &\quad (\forall p \in \alpha.\exists q \in \beta.P(p,q)\\ &\qquad \supset \forall x \in L(\alpha).\exists y \in L(\beta).\\ &\qquad\quad length(x) = length(y)\\ &\qquad\quad \wedge\forall i \in nat.(1 \leq i \leq length(x) \supset P(elem(i,x), elem(i,y))))\end{aligned}$$

This specification can be proved by $(\forall^2 I)$, $(\supset I)$ and $(L(\alpha)\text{-}ind)$, and a program scheme will be extracted from the proof.
Let the specification be, for simplicity, $\forall^2\alpha.\forall^2\beta.\forall^2 P.(\forall p.\exists q.P(p,q) \supset SPEC(\alpha,\beta,P))$. Application of the map function to an *even_odd* function, which is actually a specification of the function and its proof, can be described as a proof procedure. A specification of

even_odd is as follows:

$$\forall p \in nat.\exists q \in bool.((\exists x \in nat.p = 2 \cdot x \wedge q = T) \vee (\exists y \in nat.p = 2 \cdot y + 1 \wedge q = F))$$

This can be proved by (*nat-ind*). Let the specification be $\forall p.\exists q.EO(p,q)$. Then, the application is described as follows:

$$\dfrac{\Sigma_0 \quad \forall p.\exists q.EO(p,q) \qquad \dfrac{\Sigma_1 \quad \lambda(p,q).EO \qquad \dfrac{\Sigma_2}{\forall^2 P.(\forall p.\exists q.P(p,q) \supset SPEC(nat,bool,P))}(\forall^2 E)_2}{\forall p.\exists q.EO(p,q) \supset SPEC(nat,bool,\lambda(p,q).EO)}}{SPEC(nat,bool,\lambda(p,q).EO)}(\supset E)$$

where Σ_0 is a proof of the specification of *even_odd*, Σ_1 is a proof that $\lambda(p,q).EO$ is a predicate of type $nat \times bool \rightarrow prop$. Σ_2 eliminates α and β in $\forall^2\alpha.\forall^2\beta.\forall^2 P.(\forall p.\exists q.P(p,q) \supset SPEC(\alpha,\beta,P))$ by $(\forall^2 E)_1$. The whole proof can be normalized by one $\supset$-reduction and three $\forall^2$-reductions (one is for $\forall^2 P$ and others are for $\forall^2\alpha$ and $\forall^2\beta$). The normalized proof does not contain the second order rules or the predicate variables, so that a λ^{qpc} term can be extracted.

6.2 Sorting Program

Any list sorting algorithm needs a total order relation on the elements of the lists, and the order relation can be parameterized. A parameterized specification of the sorting problem is as follows.

SORT: $\forall^2\alpha \in type_1.\forall^2 P \in \alpha \times \alpha \rightarrow prop.$
$\quad (REL(P,\alpha) \supset SORTING(P,\alpha))$
where $SORTING(P,\alpha) \stackrel{\text{def}}{=} \forall x : L(\alpha).\exists y : L(\alpha).PERM(x,y) \wedge SORTED(y,P))$

$REL(P,\alpha)$ is a formula which means that P is a total order on α. $PERM(x,y)$ means that y can be obtained by permutation of x. They can be defined suitably. $SORTED(y,P)$ means that y is sorted with the order relation, P, namely,

$$SORTED(y,P) \stackrel{\text{def}}{=} \forall i \in nat.\forall j \in nat.(1 \leq i < j < length(y) \supset P(elem(i,y),elem(j,y)))$$

The specification can be proved by two applications of $(\forall^2 I)$ and $(\supset I)$ followed by, say, $(L(\sigma)\text{-}ind)$.
This specification can be applied, for example, to the following total order relation on pairs of natural numbers of type $nat \times nat$.

$$lex(x,y) \stackrel{\text{def}}{=} proj(1)(x) < proj(1)(y) \vee (proj(1)(x) = proj(1)(y) \wedge proj(2)(x) \leq proj(2)(y))$$

By substituting $nat \times nat$ and $\lambda(x,y).lex$ into α and P with $(\forall^2 E)_1$ and $(\forall^2 E)_2$, and by applying two $\forall^2$-reductions, the following first order proof is obtained:

$$\dfrac{\Pi_1 \qquad \Pi_2}{SORTING(\lambda(x,y).lex, nat \times ant)}(\supset E)$$

where Π_1 defines the sorting algorithm and Π_2 is the proof that $lex(x,y)$ is actually a total order.

$$\Pi_1 \stackrel{\text{def}}{=} \frac{\begin{array}{c}[REL(\lambda(x,y).lex, nat \times nat)] \\ \Sigma_1 \\ SORTING(\lambda(x,y).lex, nat \times nat)\end{array}}{REL(\lambda(x,y).lex, nat \times nat) \supset SORTING(\lambda(x,y).lex, nat \times nat)}(\supset I)$$

$$\Pi_2 \stackrel{\text{def}}{=} \begin{array}{c}\Sigma_2 \\ REL(\lambda(x,y).lex, nat \times nat)\end{array}$$

7 Example of Program Extraction

The code extracted from the second specification of a map-function and its application to a function will be given. For the ease of presentation, the code given here is somewhat simplified by applying, informally, the partial evaluation with regard to *let*-sentences and the primitive functions on sequences, and irrelevant parameters in the Rv-schemes are omitted.

(1) The following program scheme is extracted from a proof of the specification.

$$\begin{array}{l} MAP \equiv \Lambda P.\lambda(x_1, RV(P)_1). \\ \quad \mu(z_1, RV(P)_2). \\ \quad\quad \lambda x.if\ x = nil \\ \quad\quad\quad then\ (nil, \lambda i.any[L(P)]) \\ \quad\quad\quad else\ (ap(x_1, hd(x)) :: ap(z_1, tl(x)), \\ \quad\quad\quad\quad \lambda i.if\ i = 1 \\ \quad\quad\quad\quad\quad then\ ap(RV(P)_1, hd(x)) \\ \quad\quad\quad\quad\quad else\ ap(ap(RV(P)_2), tl(x)), pred(i))) \end{array}$$

where $(x_1, RV(P)_1)$ is the value of $Rv(\forall p.\exists q.P(p,q))$ which is extracted from the assumption $\forall p.\exists q.P(p,q)$ discharged in the application of $(\supset I)$. $(z_1, RV(P)_2)$ is the value of $Rv(H)$ where H is the induction hypothesis: $H \stackrel{\text{def}}{=} \exists y.(length(tl(x)) = length(y) \wedge \forall i.(1 \leq i \leq length(tl(x)) \supset P(elem(i, tl(x)), elem(i,y))))$. Notice that z_1 is a realing variable corresponding to $\exists y$ part of H.

If MAP is applied to a specification of a function and its proof, we obtain a recursive call function. The recursive call function will calculate a sequence of terms: the first element is the value of $y \in L(\beta)$, which is nil or $ap(x_1, hd(x)) :: ap(z_1, tl(x)))$, and the rest of the sequence is a justification of $length(x) = length(y) \wedge \forall i.(1 \leq i \leq length(x) \supset P(elem(i,x), elem(i,y)))$. The justification part is sometimes, but not always, redundant. More specifically, if a suitable predicate is substituted to P and the predicate has some computational contents, the justification part will be redundant in the obtained program.

(2) The following program is extracted from the proof describing application of the map-

function to *even_odd*. The predicate variable, P, is removed by the proof normalization with regard to $\forall^2$.

$$
\begin{array}{l}
ap(\lambda(x_1,x_2,x_3,x_4).\\
\quad \mu(z_1,z_2,z_3,z_4).\\
\quad \lambda x.if\ x = nil\\
\qquad then\ (nil, \lambda i.any[3])\\
\qquad else\ (ap(x_1, hd(x)) :: ap(z_1, tl(x)),\\
\qquad\quad \lambda i.if\ i = 1\ then\ ap((x_2,x_3,x_4), hd(x))\\
\qquad\qquad\qquad else\ ap(ap((z_2,z_3,z_4), tl(x)), pred(i))\\
\quad \mu(w_1,w_2,w_3,w_4).\\
\qquad \lambda p.if\ p = 0\\
\qquad\quad then\ (T, left, 0, any[1])\\
\qquad\quad else\ if\ ap(w_2, pred(p)) = left\\
\qquad\qquad then\ (F, right, any[1], ap(w_3, pred(p)))\\
\qquad\qquad else\ (T, left, pred(ap(w_4, pred(p))), any[1])
\end{array}
$$

Notice that $RV(P)_1$, $RV(P)_2$, and $any[L(P)]$ are instantiated to (x_2,x_3,x_4), (z_2,z_3,z_4), and $any[3]$.

The function, $\mu(w_1,w_2,w_3,w_4).\lambda p.\cdots$, is the code extracted from a proof of *even_odd*. The top level application, $ap(\_,\_)$, corresponds to the $(\supset E)$ application. If $\supset$-reduction is performed before the code extraction, the obtained code is as follows:

$$
\begin{array}{l}
\mu(z_1,z_2,z_3,z_4).\ \lambda x.if\ x = nil\\
\qquad\qquad\qquad then\ (nil, \lambda i.any[3])\\
\qquad\qquad\qquad else\ let\ (y_1,y_2,y_3,y_4)\\
\qquad\qquad\qquad\qquad = ap(\mu(w_1,w_2,w_3,w_4)\\
\qquad\qquad\qquad\qquad\qquad \lambda p.if\ p = 0\\
\qquad\qquad\qquad\qquad\qquad\quad then\ (T, left, 0, any[1])\\
\qquad\qquad\qquad\qquad\qquad\quad else\ if\ ap(w_2, pred(p)) = left\\
\qquad\qquad\qquad\qquad\qquad\qquad then\ (F, right, any[1], ap(w_3, pred(p)))\\
\qquad\qquad\qquad\qquad\qquad\qquad else\ (T, left, pred(ap(w_4, pred(p))), any[1]),\\
\qquad\qquad\qquad\qquad\qquad hd(x))\\
\qquad\qquad\qquad in\ (y_1 :: ap(z_1, tl(x)),\\
\qquad\qquad\qquad\qquad \lambda i.\ if\ i = 1\\
\qquad\qquad\qquad\qquad\quad then\ (y_2,y_3,y_4)\\
\qquad\qquad\qquad\qquad\quad else\ ap(ap((z_2,z_3,z_4), tl(x)), pred(i)))
\end{array}
$$

This program has the redundant justification part because of the redundant computational contents in *even_odd*. It can be removed by using the extended projection method. Then, the final code is as follows:

$$
\begin{array}{l}
\mu z_1.\lambda x.if\ x = nil\\
\qquad then\ nil\\
\qquad else\ let\ (y_1,y_2)\\
\qquad\qquad = ap(\mu(w_1,w_2).
\end{array}
$$

$$\lambda p.if\ p = 0\ then\ (T, left)$$
$$else\ if\ ap(w_2, pred(p)) = left\ then\ (F, right)$$
$$else\ (T, left),$$
$$hd(x))$$
$$in\ y_1 :: ap(z_1, tl(x))$$

(3) Application of the map function defined in 6.1 (2) to a function can also be carried out at the program scheme level. For example, assume again the *even_odd* function. First apply MAP to the predicate $\lambda(p,q).((\exists x \in nat.p = 2 \cdot x \wedge q = T) \vee (\exists y \in nat.p = 2 \cdot x + 1 \wedge q = F))$. Then, appling the reduction rule on the program schemes, the following specialized map-function is obtained:

$$\lambda(x_1, w_1, w_2, w_3).$$
$$\mu(z_1, w_1', w_2', w_3').$$
$$\lambda x.if\ x = nil$$
$$then\ (nil, \lambda i.any[3])$$
$$else\ (ap(x_1, hd(x)) :: ap(z_1, tl(x)),$$
$$\lambda i.if\ i = 1$$
$$then\ ap((w_1, w_2, w_3), hd(x))$$
$$else\ ap(ap((w_1', w_2', w_3'), tl(x)),$$
$$pred(i)))$$

Finally, we apply this program to the *even_odd* function extracted from a proof of the specification.

Notice that since the extended projection method can be used at the proof description level, the code obtained by applying MAP at the proof level is much better than the code obtained here.

8 Conclusion and Discussion

We have proposed a second order constructive calculus called $\mathbf{QPC}_2$. It does not use Prawitz coding of logical connectives since we aim to synthesize natural programs with the primitive constructs such as if-then-else and pairing. Our system is roughly a restriction of Martin-Löf's ITT with the universes U_1 and U_2 in the type free formalism, but, unlike ITT, universal quantification over predicates is allowed. The second order formulas being suitably restricted, automatic instantiation of parameterized specifications is possible. A version of our system was implemented as the SHUTEN system [20] and experimental studies have been carried out. The calculus seems to be almost sufficient for describing the parameterized specifications.

(1) Redundancy in program schemes
$\mathbf{QPC}_2$ does not have, except Harrop formulas [22], the notion of *non-informative proposition* as seen in [14] from which no computational contents is extracted. Therefore, the

extracted sorting programs from the proofs given in section 6.2 can contain redundant code: the code, M, extracted from the proof of $(\Sigma_2/REL(\lambda(x,y).lex, nat \times nat))$. If M is not an empty code, it is redundant because the code is irrelevant to the sorting procedure. This redundancy can be removed by applying the extended projection method after the instantiation of the parameters. It will be also possible to remove the redundancy without instantiating the parameters if we introduce two kinds of constants *prop* (non-informative proposition) and *spec* (informative proposition) as in [14], and modify qpc-realizability as follows:

8) e qpc $\forall^2 P \in \mathcal{K}_1.A \stackrel{\text{def}}{=} \forall^2 P \in \mathcal{K}_1.(e \text{ qpc } A)$

9) e qpc $\forall^2 P \in \mathcal{K}_2.A \stackrel{\text{def}}{=} \forall^2 P \in \mathcal{K}_2.(ap(e,P) \text{ qpc } A)$

10) e qpc $\forall^2 \alpha \in type_1.A \stackrel{\text{def}}{=} \forall^2 \alpha \in type_1.(e \text{ qpc } A)$

where $\mathcal{K}_1 = prop$ or $\sigma \to prop$ and $\mathcal{K}_2 = spec$ or $\sigma \to spec$.

(2) Possible extensions

To describe parameterized module specifications, a second order existential quantifier corresponding to the strong sum should be introduced. Also, since the type structure and the higher order formulas of our calculus are restricted, we cannot define various data structures as in the impredicative type systems such as F, F_ω and CoC. Instead, the inductive predicates definition as seen in PX, μEON [21] and SST [16] may be incorporated in our system.

(3) Type theoretic formulation

$\mathbf{QPC_2}$ is defined in a type free formalism because the extended projection method (EPM) can be directly applied. A type theoretic formulation is likely to be possible if we use the subset type technique or the informative/noninformative type technique instead of EPM. Also, a type theoretic formulation of EPM has been pursuited in the ATT system [7], so that it seems to be possible to formalize most part of the $\mathbf{QPC_2}$ as a constructive type theory.

References

[1] R. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[2] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[3] Projet FORMEL. The Calculus of Constructions, Documentation and users's guide Version 4.10. Technical Report N° 110, Institut National de Recherche en Informatique et en Automatique, 1989.

[4] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'order supérieur.* Thèse d'état. Université Paris 7, 1972.

[5] J.-Y. Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45, 1986.

[6] C. A. Goad. *Computational Uses of the Manipulation of Formal Proofs.* PhD thesis, Stanford University, 1980.

[7] S. Hayashi. Singleton, Union and Intersection Types for Program Extraction. In *Proceedings of First International Conference on Theoretical Aspects of Computer Software, LNCS 526.* Springer–Verlag, 1991.

[8] S. Hayashi and H. Nakano. *PX : A Computational Logic.* The MIT Press, 1989.

[9] G. Kreisel and A. S. Troelstra. Formal systems for some branches of intuitionistic analysis. *Annals of Mathematical Logic*, 1, 1970.

[10] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[11] C. Mohring. Algorithm Development in the Calculus of Constructions. In *Proceedings of Symposium on Logic in Conputer Science.* IEEE, 1986.

[12] B. Nordström and K. Petersson. Programming in constructive set theory: some examples. In *Proceedings of 1981 Conference on Functional Programming Language and Computer Architecture.* ACM, 1981.

[13] B. Nordström, K. Petersson and J. Smith. *Programming in Martin-Löf's Type Theory, An Introduction.* International Series of Monographs on Computer Science 7. Oxford Science Publications, 1990.

[14] C. Paulin-Mohring. Extracting F_ω's Programs from Proofs in the Calculus of Constructions. In *16th Annual ACM Symposium on Principles of Programming Languages.* ACM, 1989.

[15] D. Prawitz. *Natural Deduction.* Almqvist and Wiksell, Stockholm, 1965.

[16] M. Sato and Y. Kameyama. Constructive Programming in SST. In *Proceedings of the Japanese-Czechoslovak Seminar on Theoretical Foundations of Knowledge Information Processing.* INORGA, 1990.

[17] Y. Takayama. QPC : QJ-Based Proof Compiler – Simple Examples and Analysis. In *European Symposium on Programming '88, LNCS 300.* Springer–Verlag, 1988.

[18] Y. Takayama. Extended Projection – a new method to extract efficient programs from constructive proofs. In *Proceedings of 1989 Conference on Functional Programming Languages and Computer Architecture.* ACM, 1989.

[19] Y. Takayama. Extraction of Redundancy-free Programs from Constructive Natural Deduction Proofs. *Journal of Symbolic Computation*, 12(1), 1991.

[20] Y. Takayama. SHUTEN: A Constructive Programming System. *Journal of Japanese Society for Artificial Intelligence*, 7(4), 1992.

[21] M. Tatsuta. Program Synthesis Using Realizability. *Theoretical Computer Science*, 90, 1991.

[22] A. S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* Lecture Note in Mathematics. 344. Springer–Verlag, 1973.

Appendix: The proof of soundness of qpc-realizability

Before proving the theorem, we must introduce a new construct $l(A)$ for a formula A.

Def. 15: $l(A)$
Assume that A is any class 2 formula. Then, the length of A, $l(A)$, is the length of $Rv(A)$ as a sequence of variables.

Notice that $l(A)$ defined above is different from $len(A)$ in **Def. 9**. The difference is seen in the treatment of predicate variables. $len(A) = l(A)$ if A does not contain any predicate variable. But $l(P(M_1, \cdots, M_n)) = 1$ while $len(P(M_1, \cdots, M_n)) = L(P(M_1, \cdots, M_n))$ for a predicate variable P. This implies that the value of $l(A)$ is always a natural number but the value of $len(A)$ may contain $L(P(\overline{M}))$ type expressions.

The distinction between l and len has a highly technical reason originated from the introduction of the notion of program schemes. The realizer of a formula $P(\overline{M})$, where P is a predicate variable, is $RV(P(\overline{M}))$ and in the interpretation procedure this is regarded as a meta variable representing a sequence of variables, possibly an empty sequence. This is the meaning of $l(P(\overline{M})) = 1$. On the other hand, in handling $(\vee I)_1$, $(\vee I)_2$ and $(\perp E)$ rules, the dummy codes $any[N]$ will be assigned to some formulas and, in calculating of N, $len(\cdots)$ is used instead of $l(\cdots)$ because we should delay the calculation of the value of N until the predicate variables are instantiated. There is a possibility that a realizer contains a fragment like $let\ RV(P(\overline{M})) = any[L(P(\overline{M}))]$, but it is safe because if a first order predicate is substituted to P the length of $RV(P(\overline{M}))$ part and the length of $any[L(P(\overline{M}))]$ part will be equal.

Proof of the theorem
The proof of (1) is performed by induction on the structure of proof trees. (2) and (3) are easily checked.
If the proof tree is A, then let $e = Rv(A)$. Let R be the name of the last rule which is used in the proof tree of A. If A is atomic, then let $e = ()$.
Other cases are as follows:
case $R = (\wedge I)$: Assume that $A \equiv B \wedge C$. Let a and b be program schemes such that a **qpc** B and b **qpc** C. Then, let e be (a, b). e **qpc** A can be proved by $(\wedge I)$.
case $R = (\wedge E)_1$: Assume that $A \wedge B$ is the premise of the R application. Let a be a program scheme such that a **qpc** $A \wedge B$, then let $e = ttseq(1, l(A))(a)$. e **qpc** A can be proved by $(\wedge E)_1$.
case $R = (\wedge E)_2$: Assume that $B \wedge A$ is the premise. Let a be a program scheme such that a **qpc** $B \wedge A$, then let $e = tseq(l(B) + 1)(a)$. e **qpc** A can be proved by $(\wedge E)_2$.
case $R = (\vee I)_1$: Assume that $A \equiv B \vee C$ and B be the premise. Let a be a program scheme such that a **qpc** B, then let $e = (left, a, any[len(C)])$. e **qpc** A can be proved by $(\vee I)_1$ and $(\wedge I)$.
case $R = (\vee I)_2$; Assume that $A \equiv B \vee C$ and C be the premise. Let b be a program scheme such that b **qpc** C, then let $e = (right, any[len(B)], b)$. e **qpc** A can be proved by $(\wedge I)$ and $(\vee I)_2$.
case $R = (\vee E)$; Assume that $B \vee C$ is the first premise of the R application. Let a be a program scheme such that a **qpc** $B \vee C$, and b and c be the program scheme which

realize A as the second and the third premises of the $(\vee E)$ application. Notice that b and c may contain $Rv(B)$ and $Rv(C)$. Then let $e = if\ proj(1)(a) = left\ then\ (let\ Rv(B) = ttseq(2, l(B))(a)\ in\ b)\ else\ (let\ Rv(C) = tseq(l(B)+2)(a)\ in\ c)$. The proof of e **qpc** A can be obtained by using the proofs of b **qpc** A and c **qpc** A with $Rv(B)$ and $Rv(C)$ substituted by $ttseq(2, l(B))(a)$ and $tseq(l(B)+2)(a)$ and by $(\vee E)$ and $(= E)$.

case $R = (\supset I)$; Assume that $A \equiv B \supset C$ and C be the premise. Let a be a program scheme such that a **qpc** C. a may contains $Rv(B)$. Then, let $e = \lambda Rv(B).a$. e **qpc** A can be proved by $(\forall I)$, $(\supset I)$, $(\wedge I)$ and $(= E)$.

case $R = (\supset E)$; Assume that $B \supset A$ and B are the premises. Let a and b be program schemes such that a **qpc** $B \supset A$ and b **qpc** B. Then, let $e = ap(a, b)$. e **qpc** A can be proved by $(\supset E)$, $(\wedge E)$ and $(\forall E)$.

case $R = (\forall I)$; Assume that $A \equiv \forall x \in \sigma.B$. Let a be a program scheme such that a **qpc** A, then let $e = \lambda x.a$. e **qpc** A can be proved by $(\forall I)$ and $(= E)$.

case $R = (\forall E)$; Assume that $M : \sigma$ and $\forall x \in \sigma.B(x)$ are the premises. Let a be a program scheme such that a **qpc** $\forall x \in \sigma.B(x)$. Then, let $e = ap(a, M)$. e **qpc** A can be proved by $(\forall E)$.

case $R = (\exists I)$; Assume that $M : \sigma$ and $B(M)$ be the premises. Let a be a program scheme such that a **qpc** $B(M)$. Then, let $e = (M, a)$. e **qpc** A can be proved by $(\wedge I)$.

case $R = (\exists E)$; Assume that $\exists x \in \sigma.B(x)$ is the first premise. Let a be a program scheme such that a **qpc** $\exists x \in \sigma.B(x)$ and b be a program scheme which realizes A as the second premise. Then, let $e = let\ (x, Rv(B)) = a\ in\ b$. The proof of e **qpc** A can be obtained from the proof of b **qpc** A by substituting $(proj(1)(a), tseq(2)(a))$ to $(x, Rv(B))$, and replacing the occurrences of $proj(1)(a) : \sigma$ and $tseq(1)(a)$ **qpc** $B(x)$ as the discharged hypotheses by the proof of a **qpc** $\exists x \in \sigma.B(x)$ followed by $(\wedge E)$.

case $R = (= E)$; Assume that $A \equiv A_x[N]$ and that $M = N$ and $A_x[M]$ be the premises. Let a be a program scheme such that a **qpc** $A_x[M]$. Then, let $e = a$. e **qpc** A can be proved by $(= E)$.

case $R = (\perp E)$; Let $e = any[len(A)]$. e **qpc** A can be proved by $(\perp E)$.

case $R = (L(\sigma)\text{-}ind)$; Assume $A \equiv \forall x \in L(\sigma).B(x)$, and A is proved as follows:

$$\frac{\begin{array}{c}[x = nil \vee (x \neq nil \wedge B(tl(x)))] \\ \Sigma \\ B(x)\end{array}}{\forall x \in L(\sigma).B(x)}(L(\sigma)\text{-}ind)$$

Let $H \stackrel{\text{def}}{=} x = nil \vee H'$ where $H' \stackrel{\text{def}}{=} x \neq nil \wedge B(tl(x))$. By the induction hypothesis, the following proof can be obtained from $(\Sigma / B(x))$ for some program scheme, e.

$$\begin{array}{c}\Sigma' \\ e\ \textbf{qpc}\ B(x)\end{array}$$

Σ' may contain the occurrences of $Rv(H)$ **qpc** $H \equiv (z = left \wedge x = nil) \vee (z = right \wedge x \neq nil \wedge B(tl(x)) \wedge \overline{x}$ **qpc** $B(tl(x)))$ where $(z, \overline{x}) \stackrel{\text{def}}{=} Rv(H)$. Let $H_1 \stackrel{\text{def}}{=} z = left \wedge x = nil$ and $H_2 \stackrel{\text{def}}{=} z = right \wedge x \neq nil \wedge B(tl(x)) \wedge \overline{x}$ **qpc** $B(tl(x))$. Let $W \stackrel{\text{def}}{=} W_1 \vee W_2 \stackrel{\text{def}}{=} x = nil \vee (x \neq nil \wedge B(tl(x)) \wedge \overline{x}$ **qpc** $B(tl(x)))$ and let $d \stackrel{\text{def}}{=} if\ x = nil\ then\ left\ else\ right$.

Now construct the following proofs:

$$\Pi_1 \stackrel{\text{def}}{\equiv} \dfrac{\dfrac{\dfrac{W_1}{\dfrac{H}{\Sigma}}(\vee I) \quad \dfrac{W_1}{d = left} \quad \dfrac{\dfrac{\dfrac{2 : type_1 \quad left : 2}{W_1 \quad left = left}(\wedge I)}{(H_1)_z[left]}(\vee I)}{(Rv(H)\ \mathbf{qpc}\ H)_z[left]} \quad \dfrac{\Sigma'_z[left]}{e_z[left]\ \mathbf{qpc}\ B(x)}(= E)}{e_z[d]\ \mathbf{qpc}\ B(x)} \quad B(x)}{B(x) \wedge e_z[d]\ \mathbf{qpc}\ B(x)}(\wedge I)$$

$$\Pi_2 \stackrel{\text{def}}{\equiv} \dfrac{\dfrac{\dfrac{\dfrac{W_2}{H'}(\wedge E)}{\dfrac{H}{\Sigma}}(\vee I) \quad \dfrac{W_2}{d = right} \quad \dfrac{\dfrac{\dfrac{2 : type_1 \quad right : 2}{W_2 \quad right = right}(\wedge I)}{(H_2)_z[right]}(\vee I)}{(Rv(H)\ \mathbf{qpc}\ H)_z[right]} \quad \dfrac{\Sigma'_z[right]}{e_z[right]\ \mathbf{qpc}\ B(x)}(= E)}{e_z[d]\ \mathbf{qpc}\ B(x)} \quad B(x)}{B(x) \wedge e_z[d]\ \mathbf{qpc}\ B(x)}(\wedge I)$$

Notice that the above proofs contain free occurrences of $\overline{x}(= Rv(B(tl(x))))$. Let $f \stackrel{\text{def}}{\equiv} \mu\overline{z}.\lambda x.e_{z,\overline{x}}[d, ap(\overline{z}, tl(x))]$ where $\overline{z}$ is a sequence of fresh variables of the same length as $\overline{x}$. By substituting $ap(f, tl(x))$ to $\overline{x}$, the following proof is constructed:

$$\dfrac{\dfrac{[W_1 \vee W_{2\overline{x}}[ap(f,tl(x))]] \quad \Pi_{1\overline{x}}[ap(f,tl(x))] \quad \begin{matrix}[W_1] \\ \Pi_{2\overline{x}}[ap(f,tl(x))]\end{matrix} \quad [W_{2\overline{x}}[ap(f,tl(x))]]}{B(x) \wedge e_{z,\overline{x}}[d, ap(f,tl(x))]\ \mathbf{qpc}\ B(x)}(\vee E)}{\forall x \in L(\sigma).B(x) \wedge e_{z,\overline{x}}[d, ap(f,tl(x))]\ \mathbf{qpc}\ B(x)}(L(\sigma)\text{-}ind)$$

From this proof, a proof of $\forall x \in L(\sigma).e_{z,\overline{x}}[d, ap(f, tl(x))]\ q\ B(x)$ can easily be obtained. This implies, from the definition of qpc-realizability and the reduction rule for μ-terms, that a proof of f **qpc** $\forall x \in L(\sigma).B(x)$ is obtained.

case $R = (\forall^2 I)$:

(1) $A \equiv \forall^2 P \in \mathcal{K}.B$ (K = *prop* or $\sigma \rightarrow prop$)

Assume that the proof is as follows:

$$\dfrac{\begin{matrix}[P : \mathcal{K}] \\ \Sigma \\ B\end{matrix}}{\forall^2 P \in \mathcal{K}.B}(\forall^2 I)$$

Let a be the program scheme such that a **qpc** B whose proof is constructed from the subproof (Σ/B). Then, $\Lambda P.a$ **qpc** $\forall^2 P \in \mathcal{K}.B \stackrel{\text{def}}{\equiv} \forall^2 P \in \mathcal{K}.(ap(\Lambda P.a, P)\ \mathbf{qpc}\ B) = \forall^2 P \in \mathcal{K}.(a\ \mathbf{qpc}\ B)$. Therefore, let $e = \Lambda P.a.\ e$ **qpc** A can be proved by $(\forall^2 I)$.

(2) $A \equiv \forall^2 \alpha \in type_1.B$

Assume that the proof is as follows:

$$\dfrac{\begin{array}{c}[\alpha : type_1]\\ \Sigma\\ B\end{array}}{\forall^2\alpha \in type_1.B}(\forall^2 I)$$

Let a be a program scheme such that a **qpc** B. Notice that by the definition of **qpc**-realizability, a **qpc** $\forall^2\alpha.B = \forall^2\alpha.a$ **qpc** B. Then let $e = a.\ e$ **qpc** A can be proved by $(\forall^2 I)$.

case $R = (\forall^2 E)_1$

(1) Universal type quantification

Assume that the proof is as follows:

$$\dfrac{\begin{array}{cc}\Sigma_0 & \Sigma_1\\ \sigma : type_1 & \forall^2\alpha \in type_1.A\end{array}}{A_\alpha[\sigma]}(\forall^2 E)_1$$

By the induction hypothesis, the following proofs are obtained:

$$\begin{array}{c}\Sigma_1'\\ a \text{ qpc } \forall^2\alpha \in type_1.A\end{array}$$

Then let $e = a.\ e$ **qpc** $A_\alpha[\sigma]$ can be proved by the $(\forall^2 E)_1$ rule.

(2) Universal proposition quantification

In this subcase, the second premise of the $(\forall^2 E)_1$ application is in the form of $\forall^2 P \in prop.B$, and the proof is carried out in the same way as in the next case.

case $R = (\forall^2 E)_2$: Assume that $A \equiv B_p[\lambda\overline{x}.p]$ and that the proof is as follows:

$$\dfrac{\begin{array}{cc}\Sigma_0 & \Sigma_1\\ \lambda\overline{x}.p \in \sigma \to prop & \forall^2 P \in \sigma \to prop.B\end{array}}{B_P[\lambda\overline{x}.p]}(\forall^2 E)_2$$

By the induction hypothesis, the following proof is obtained for some program scheme, a:

$$\begin{array}{c}\Sigma_1'\\ a \text{ qpc } \forall^2 P \in \sigma \to prop.B\end{array}$$

Then let $e = ap(a, \lambda\overline{x}.p).\ e$ **qpc** $B_P[\lambda\overline{x}.p]$ can be proved by the $(\forall^2 E)_2$ rule. ∎